Analysing Your Defect Data for Improvement Potential

Otto Vinter,
Independent Sw.,
Denmark
Analysing Your Defect Data for Improvement Potential

Otto Vinter
Software Engineering Mentor
Tel/Fax: +45 4399 2662, Mobile: +45 4045 0771
vinter@inet.uni2.dk
www.vinter.suite.dk
Why Defect Analysis?

Objective
- identify why defects occur
- help prevent defect insertion

Assumption
- defect detection costs both time and money
- defects cost more when detected later in the life cycle
- fewer defects can be inserted through training/education

Strategy
- move defect detection to earlier parts of the life cycle
- avoid defect insertion by improving the process
Defect Analysis Models

Mathematical models

- quantitative analysis
- statistics
- growth curve modelling
- comparison to historical data
- not easily translatable to corrective action

Causal analysis

- qualitative analysis
- investigate details of a sample of defects
- fewer restrictions on what can be analysed
- time consuming
- harder to aggregate
- easy to translate to individual improvement actions
Defect Analysis in Practice

Why mathematical models often fail to predict

- low-level maturity processes are the norm
- software development is not a production process
- personal and team issues dominate

Why causal analysis often works

- focuses on prevention
- addresses individual and team issues
- feedback, education, motivation
- bottom-up process improvement
Tools for Performing Defect Analysis

For classifying defects
- Boris Beizer’s bug taxonomy

For capturing quantitative information
- When found: distribution over time (before/after release)
- Where found: distribution across system parts (subsystems/modules)
- Who found: distribution across stakeholder groups (project/specialists/customers)

For capturing qualitative information
- Interviews with the relevant developers
- What happened: listen and learn (problem/explanation/insight/cause)
- How to prevent: ad-hoc check lists (literature/history/experience)
Retrospective Defect Analysis

Retrospective analysis

- combined bug classification and qualitative analysis
- performed on a sample of at least 100 bug reports
- investigates potential prevention techniques
- proposes focused improvement actions

Interview sessions

- 1-2 developers and 1-2 process consultants
- app. 4 min / bug for classification and qualitative information
- app. 30 minutes / bug for detailed analysis of selected bugs to determine potentials for improvement (prevention)
When Are Defects Found (example)
Defect Detection Percentage Over Time (example)

\[ DDP(t) = \frac{\text{Defects before release}}{\text{All defects (t)}} \times 100 \% \]
Where / Who Found the Defects (example)
Boris Beizer’s Bug Taxonomy

Bug classification

- nine main categories
- detailed in up to four levels
- open and extendable

Statistics

- Boris Beizer
  - 16,209 bugs in Assembler, FORTRAN, COBOL, ADA, C
  - primarily US defence, aerospace, and telecommunication
  - collected before 1989
- Otto Vinter
  - 982 bugs in Pascal, C, C++
  - embedded and PC applications at Brüel & Kjær, DK
  - collected from 1992 – 1998

More information: www.vinter.suite.dk/engdefana.htm
The Beizer Bug Taxonomy

1. Requirements and Features
2. Functionality as Implemented
3. Structural Bugs
4. Data
5. Implementation
6. Integration
7. System and Software Architecture
8. Test Definition or Execution Bugs
9. Other Bugs, Unspecified
Bug Taxonomy Notation

Each category is detailed to a depth of 4+ levels

- 3xxx -- structural bugs in the implemented software
- 32xx -- processing bugs
- 322x -- expression evaluation bugs
- 3222 -- arithmetic expressions bugs
- 3222.1 -- wrong operator bugs

The "x" is a place holder for possible future filling in of numbers as the taxonomy is (inevitably) expanded.

Categories ending on “9” are used when a finer breakdown is not available, e.g. for bugs that do not fit in the other (sub)categories:

- 9xxx -- other or unspecified bugs
- 39xx -- other structural bugs
- 329x -- other processing bugs
- 3229 -- other expression evaluation bugs
- 3222.9 -- other arithmetic bugs
B&K Top 12 Beizer Categories (52% of all bugs)

- 211x Feature misunderstood, Wrong: 8%
- 231x Missing cases: 7%
- 9xxx Other bugs, unspecified: 6%
- 218x Feature interactions: 5%
- 324x Cleanup: 4%
- 323x Initialization: 3%
- 131x Incomplete requirements: 3%
- 132x Missing, unspecified requirement: 3%
- 1621 Feature Added: 3%
- 413x Data definition initial, default values: 3%
- 241x Domain misunderstood, Wrong: 3%
- 8111 Requirements misunderstood in test design: 3%
Top Prevention Techniques (expensive example)
Top Prevention Categories (simpler/cheaper example)

- Better analysis of consequences: 20%
- Better analysis of requirements: 18%
- Improved code review: 12%
- Enforce coding standards: 10%
- Unknown: 8%
1st Improvement Action: The Prevention of Errors through Experience-driven Test Efforts (PET)

Results of the analysis of bug reports

- no special bug class dominates embedded software development
- requirements problems, and requirements related problems, are the prime bug cause (>36%)
- problems due to lack of systematic unit testing is the second largest bug cause (22%)

Action: Improve Testing Processes

- static analysis
  - source code complexity, data flow anomalies, coding standards
- dynamic analysis
  - code coverage by test cases, cross references code to test cases

Funded by CEC. More information: www.vinter_suite.dk/engswtest.htm
Results of the Improved Testing Process

46% Improvement in testing efficiency
- removing static bugs
- increasing unit branch coverage to > 85%

75% Reduction in production-release bugs
- Compared to trial-release

70% Requirements bugs in production-release

Next action: Improve Requirements Engineering
Results of the analysis of requirements bugs

- requirements related bugs: 51%
- usability issues dominate: 64%
- external software issues: 28%
- functionality issues: 22%

Action: Improve Requirements Elicitation and Validation

- Scenarios
  - Relate demands to use situations. Describe tasks for each scenario.
- Usability Test
  - Check that the users are able to use the system for daily tasks, based on a navigational prototype of the user interface.

Funded by CEC. More information: www.vinter.suite.dk/engreqeng.htm
Results of the 2nd Improvement Action

Product sold more than twice as many copies in the first year
- compared to a similar product developed for a similar domain by the same team

Usability was exceptional in the market
- users’ interaction with the product was totally changed as a result of the early usability tests

Almost 3 times increase in productivity for the development of the user interface

27% reduction in problem reports
When the benefits of performing a defect analysis are so evident, why don’t you start analysing your defect data?

Thank you for listening

Otto Vinter
Software Engineering Mentor
Tel/Fax: +45 4399 2662, Mobile: +45 4045 0771
vinter@inet.uni2.dk
www.vinter.suite.dk